

# Knight Moves — BFS Approach on a Chessboard

The Knight Moves problem asks for the fewest moves a knight needs to travel from one square to another on a regular 8×8 chessboard. Since a knight always moves in an L-shape—either two squares in one direction and then one square sideways, or the reverse—each square typically has up to eight reachable positions unless the board edges limit them. Input consists of multiple test cases, each containing two positions in algebraic chess notation (a1–h8). For every test case, the program must output:

“To get from [start] to [end] takes [n] knight moves.”

Because all knight moves have equal weight, the chessboard can be modeled as an unweighted graph: each square represents a node, and each legal knight move represents an edge.

Therefore, a Breadth-First Search (BFS) is the ideal method—it explores moves level by level and guarantees the shortest path in terms of move count. The algorithm converts algebraic square names into numeric coordinates, validates each move to ensure it stays inside the board, maintains a visited list to avoid revisiting squares, and uses a queue to manage BFS progression.

## Step-by-Step Breakdown

### Step 1: Problem Insight

We must compute the minimum number of L-shaped knight jumps required to travel from the starting square to the target square. A knight may move in eight possible ways, combining two steps in one axis and one in the other.

### Step 2: Chessboard Representation

Model the board as an 8×8 grid.

Algebraic notation conversion:

- Columns: 'a' to 'h' → 0–7
- Rows: '1' to '8' → 0–7

So, for example, “e2” becomes (4, 1).

### Step 3: Knight Movement Set

All the knight’s possible displacements can be represented with two arrays:

- $dx = \{ 2, 2, 1, 1, -1, -1, -2, -2 \}$
- $dy = \{ 1, -1, 2, -2, 2, -2, 1, -1 \}$

Each pair ( $dx[i]$ ,  $dy[i]$ ) gives one valid knight jump direction.

#### Step 4: BFS Initialization

To run BFS efficiently:

- A visited[8][8] array keeps track of explored cells.
- Optionally, a distance array may store move counts.
- A queue is used to traverse positions in increasing move order.
- A small structure/class can hold the coordinates and the number of moves taken to reach that position.

#### Step 5: Early Exit Condition

If the start square and destination square are identical, the answer is simply 0 moves.

#### Step 6: Running BFS

The BFS logic proceeds as follows:

1. Push the starting square into the queue with move count 0.
2. Mark it as visited.
3. While the queue still has items:
  - Remove the front element.
  - Try all eight knight move offsets.
  - For each candidate square:
    - Check if it stays within 0–7 boundary.
    - Ignore if already visited.
    - If it matches the target square → return moves + 1 immediately.
    - Otherwise, mark visited and push it into the queue with incremented move count.

This guarantees the first time we reach the destination is via the shortest path.

#### Step 7: Input Handling and Output Formatting

- Read pairs of algebraic squares until EOF.
- For every pair, compute the knight's minimum distance using BFS.

- Output strictly in the required format:  
"To get from xx to yy takes n knight moves."

#### Step 8: Valid Move Check

Always ensure that a coordinate (x, y) is legal by verifying:

$$0 \leq x < 8$$

$$0 \leq y < 8$$

Only moves satisfying this condition can be used during BFS.

Input:

e2 e4

a1 b2

b2 c3

a1 h8

a1 h7

h8 a1

b1 c3

f6 f6

Execution:

Input: e2 e4

1. Read Input: start = "e2", end = "e4"

2. Convert Coordinates:

o e2 → column 'e' = 4, row '2' = 1 → (4,1) o e4 → column 'e' = 4, row  
'4' = 3 → (4,3)

3. Initialize BFS:

o Create visited[8][8] = all false o Queue = [(4,1,0)] o visited[4][1] =

true

4. Process Queue:

- o Dequeue (4,1,0)
- o Generate 8 knight moves: (6,2), (6,0), (5,3), (5,-1), (3,3), (3,-1), (2,2), (2,0)
- o Valid moves (within board): (6,2), (6,0), (5,3), (3,3), (2,2), (2,0)
- o Enqueue all valid moves with moves=1

5. Continue BFS:

- o Dequeue (6,2,1)
  - o Generate moves: (8,3), (8,1), (7,4), (7,0), (5,4), (5,0), (4,3), (4,1) o
- Found destination (4,3) → return 1 + 1 = 2

6. Output: "To get from e2 to e4 takes 2 knight moves."

Input: a1 b2

1. Read Input: start = "a1", end = "b2"
2. Convert Coordinates: a1 → (0,0), b2 → (1,1)
3. BFS Search: Explores multiple paths through intermediate squares
4. Find Path: (0,0)→(2,1)→(1,3)→(3,2)→(1,1) = 4 moves
5. Output: "To get from a1 to b2 takes 4 knight moves."

Input: b2 c3

1. Read Input: start = "b2", end = "c3"
2. Convert Coordinates: b2 → (1,1), c3 → (2,2)
3. BFS Search: Finds path through one intermediate square
4. Find Path: (1,1)→(2,3)→(2,2) = 2 moves
5. Output: "To get from b2 to c3 takes 2 knight moves."

Input: a1 h8

1. Read Input: start = "a1", end = "h8"
2. Convert Coordinates: a1 → (0,0), h8 → (7,7)
3. BFS Search: Explores longest diagonal path
4. Find Path: Multiple 6-move paths exist across board
5. Output: "To get from a1 to h8 takes 6 knight moves."

Input: a1 h7

1. Read Input: start = "a1", end = "h7"
2. Convert Coordinates: a1 → (0,0), h7 → (7,6)
3. BFS Search: Slightly shorter than h8 path
4. Find Path: Optimal path found in 5 moves
5. Output: "To get from a1 to h7 takes 5 knight moves."

Input: h8 a1

1. Read Input: start = "h8", end = "a1"
2. Convert Coordinates: h8 → (7,7), a1 → (0,0)
3. BFS Search: Reverse of previous path
4. Find Path: Same 6 moves as forward direction
5. Output: "To get from h8 to a1 takes 6 knight moves."

Input: b1 c3

1. Read Input: start = "b1", end = "c3"
2. Convert Coordinates: b1 → (1,0), c3 → (2,2)
3. BFS Search: Direct L-shaped move available
4. Find Path: (1,0)→(2,2) = 1 move
5. Output: "To get from b1 to c3 takes 1 knight moves."

Input: f6 f6

1. Read Input: start = "f6", end = "f6"

2. Convert Coordinates: f6 → (5,5), f6 → (5,5)
3. Special Case: Same start and end position
4. Immediate Return: 0 moves without BFS
5. Output: "To get from f6 to f6 takes 0 knight moves."

Output:

To get from e2 to e4 takes 2 knight moves.

To get from a1 to b2 takes 4 knight moves.

To get from b2 to c3 takes 2 knight moves.

To get from a1 to h8 takes 6 knight moves.

To get from a1 to h7 takes 5 knight moves.

To get from h8 to a1 takes 6 knight moves.

To get from b1 to c3 takes 1 knight moves.

To get from f6 to f6 takes 0 knight moves.

**Pseudocode:** TYPE Node:

row

col

stepCount

MOVE\_ROW = [ 2, 2, 1, 1, -1, -1, -2, -2 ]

MOVE\_COL = [ 1, -1, 2, -2, 2, -2, 1, -1 ]

FUNCTION insideBoard(r, c):

RETURN (0 ≤ r < 8) AND (0 ≤ c < 8)

FUNCTION knightBFS(startSq, targetSq):

IF startSq = targetSq:

RETURN 0

DECLARE visited[8][8] initialized to FALSE

startR = startSq[0] - 'a'

startC = startSq[1] - '1'

goalR = targetSq[0] - 'a'

goalC = targetSq[1] - '1'

CREATE queue Q

ENQUEUE Q with Node(startR, startC, 0)

visited[startR][startC] = TRUE

WHILE Q NOT EMPTY:

now = Q.FRONT()

Q.DEQUEUE()

FOR k FROM 0 TO 7:

nr = now.row + MOVE\_ROW[k]

nc = now.col + MOVE\_COL[k]

IF insideBoard(nr, nc) AND visited[nr][nc] = FALSE:

```
IF nr = goalR AND nc = goalC:  
    RETURN now.stepCount + 1  
  
visited[nr][nc] = TRUE  
ENQUEUE Q with Node(nr, nc, now.stepCount + 1)  
  
RETURN -1 // should never happen on an 8x8 board
```

```
FUNCTION main():  
    WHILE input has (src, dst):  
        ans = knightBFS(src, dst)  
        PRINT "To get from " + src + " to " + dst + " takes " + ans + " knight moves."
```

**Here is the solution code for Knight Moves:** <https://github.com/pulokdas062/Graph-Algorithm/blob/main/BFS/Knight%20Moves/knightmoves.cpp>